



Building Multi-File Programs with the make Tool



1

1

Agenda



Motivation for Make

Make Fundamentals
Non-File Targets
Macros

2

2

Multi-File Programs



intmath.h (interface)

```
#ifndef INTMATH_INCLUDED
#define INTMATH_INCLUDED
int gcd(int i, int j);
int lcm(int i, int j);
#endif
```

intmath.c (implementation)

```
#include "intmath.h"

int gcd(int i, int j)
{
    int temp;
    while (j != 0)
    {
        temp = i % j;
        i = j;
        j = temp;
    }
    return i;
}

int lcm(int i, int j)
{
    return (i / gcd(i, j)) * j;
}
```

testintmath.c (client)

```
#include "intmath.h"
#include <stdio.h>

int main(void)
{
    int i;
    int j;
    printf("Enter the first integer:\n");
    scanf("%d", &i);
    printf("Enter the second integer:\n");
    scanf("%d", &j);
    printf("Greatest common divisor: %d.\n",
        gcd(i, j));
    printf("Least common multiple: %d.\n",
        lcm(i, j));
    return 0;
}
```

Note: intmath.h is
#included into intmath.c
and testintmath.c

3

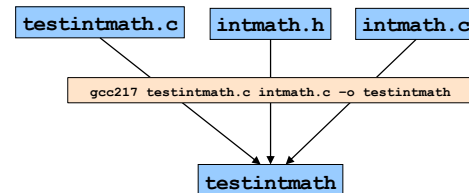
3

Motivation for Make (Part 1)



Building testintmath, approach 1:

- Use one gcc217 command to preprocess, compile, assemble, and link



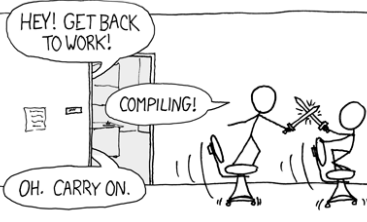
4

4

<https://xkcd.com/303/>



THE #1 PROGRAMMER EXCUSE
FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."



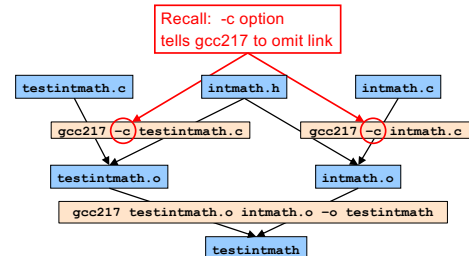
5

Motivation for Make (Part 2)



Building testintmath, approach 2:

- Preprocess, compile, assemble to produce .o files
- Link to produce executable binary file



6

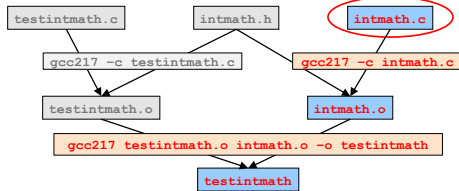
6

Partial Builds

Approach 2 allows for **partial builds**

- Example: Change `intmath.c`
 - Must rebuild `intmath.o` and `testintmath`
 - Need not rebuild `testintmath.o`

If program contains many files, could save many hours of build time



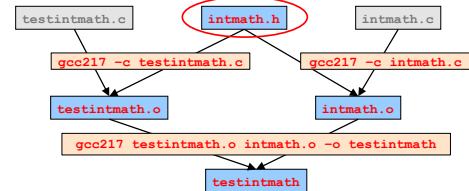
7

7

Partial Builds

However, changing a `.h` file can be more dramatic

- Example: Change `intmath.h`
 - `intmath.h` is `#included` into `testintmath.c` and `intmath.c`
 - Must rebuild `testintmath.o`, `intmath.o`, and `testintmath`



8

8

Wouldn't It Be Nice If...

Observation

- Doing partial builds manually is tedious and error-prone
- Wouldn't it be nice if there were a tool...

How would the tool work?

- Input:
 - Dependency graph (as shown previously)
 - Specifies file dependencies
 - Specifies commands to build each file from its dependents
 - Date/time stamps of files
- Algorithm:
 - If file B depends on A **and** date/time stamp of A is newer than date/time stamp of B, **then** rebuild B using the specified command

That's **make**!

9

9

Agenda

Motivation for Make

Make Fundamentals

Non-File Targets

Macros

10

10

Make Command Syntax

Command syntax

```
$ man make
```

SYNOPSIS

```
make [-f makefile] [options] [targets]
```

- makefile**
 - Textual representation of dependency graph
 - Contains **dependency rules**
 - Default name is `makefile`, then `Makefile`
- target**
 - What `make` should build
 - Usually: `.o` file, or an executable binary file
 - Default is first one defined in `makefile`

11

11

Dependency Rules in Makefile

Dependency rule syntax

```
target: prerequisites
<tab>command
```

- target**: the file you want to build
- dependencies (aka prerequisites)**: the files needed to build the target
- command**: what to execute to build the target

Dependency rule semantics

- Build **target** iff it is older than any of its **dependencies**
- Use **command** to do the build

Work recursively; examples illustrate...

12

12

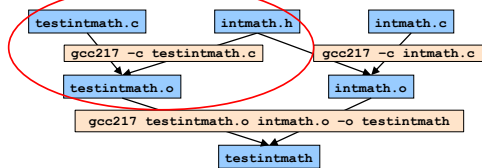
Makefile Version 1

Makefile:

```
testintmath: testintmath.o intmath.o
gcc217 testintmath.o intmath.o -o testintmath

testintmath.o: testintmath.c intmath.h
gcc217 -c testintmath.c

intmath.o: intmath.c intmath.h
gcc217 -c intmath.c
```



13

Version 1 in Action

At first, to build testintmath make issues all three gcc commands

Use the touch command to change the date/time stamp of intmath.c

```
$ make testintmath
gcc217 -c testintmath.c
gcc217 -c intmath.c
gcc217 testintmath.o intmath.o -o testintmath

$ touch intmath.c

$ make testintmath
gcc217 -c intmath.c
gcc217 testintmath.o intmath.o -o testintmath

$ make testintmath
make: 'testintmath' is up to date.

$ make
make: 'testintmath' is up to date.
```

make does a partial build

make notes that the specified target is up to date

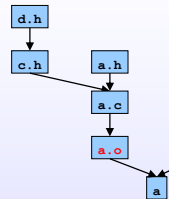
The default target is testintmath, the target of the first dependency rule

14

iClicker Question

Q: If you were making a **Makefile** for this program, what should **a.o** depend on?

- A. a
- B. a.c
- C. a.c a.h
- D. a.h c.h d.h
- E. a.c a.h c.h d.h

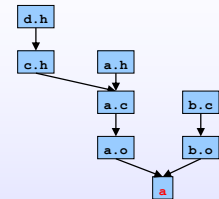


15

iClicker Question

Q: If you were making a **Makefile** for this program, what should **a** depend on?

- A. a.o b.o
- B. a.o b.o a.c b.c
- C. a.o b.o a.h c.h d.h
- D. a.c b.c a.h c.h d.h
- E. a.o b.o a.c b.c a.h c.h d.h



17

Agenda

- Motivation for Make
- Make Fundamentals
- Non-File Targets**
- Macros

19

Non-File Targets

Adding useful shortcuts for the programmer

- **make all**: create the final executable binary file(s)
- **make clean**: delete all .o files, executable binary file(s)
- **make clobber**: delete all Emacs backup files, all .o files, executable(s)

Commands in the example

- **rm -f**: remove files without querying the user
- Files ending in '~' and starting/ending in '#' are Emacs special files

```
all: testintmath
clobber: clean
rm -f *~ \#*\#
clean:
rm -f testintmath *.o
```

20

19

20

Makefile Version 2

```
# Dependency rules for non-file targets
all: testintmath
clobber: clean
rm -f *~ \#*\#
clean:
rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
gcc217 testintmath.o intmath.o -o testintmath
testintmath.o: testintmath.c intmath.h
gcc217 -c testintmath.c
intmath.o: intmath.c intmath.h
gcc217 -c intmath.c
```

21

21

Version 2 in Action

make observes that "clean" target doesn't exist; attempts to build it by issuing "rm" command

```
$ make clean
rm -f testintmath *.o

$ make clobber
rm -f testintmath *.o
rm -f *~ \#*\#

$ make all
gcc217 -c testintmath.c
gcc217 -c intmath.c
gcc217 testintmath.o intmath.o -o testintmath

$ make
make: Nothing to be done for 'all'.
```

Same idea here, but "clobber" depends upon "clean"

"all" depends upon "testintmath"

"all" is the default target

22

22

Agenda

Motivation for Make
Make Fundamentals
Non-File Targets
Macros

23

23

Macros

make has a macro facility

- Performs textual substitution
- Similar to C preprocessor's **#define**

Macro definition syntax

macroname = **macrodefinition**

- **make** replaces **\$(macroname)** with **macrodefinition** in remainder of Makefile

Example: Make it easy to change (or swap) build commands

```
CC = gcc217#m
YACC = bison -d -y
#YACC = yacc -d
```

Example: Make it easy to change build flags

```
CFLAGS = -D NDEBUG -O
```

24

24

Makefile Version 3

```
# Macros
CC = gcc217
# CC = gcc217#m
CFLAGS =
# CFLAGS = -g
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O

# Dependency rules for non-file targets
all: testintmath
clobber: clean
rm -f *~ \#*\#
clean:
rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
$(CC) $(CFLAGS) testintmath.o intmath.o -o testintmath
testintmath.o: testintmath.c intmath.h
$(CC) $(CFLAGS) -c testintmath.c
intmath.o: intmath.c intmath.h
$(CC) $(CFLAGS) -c intmath.c
```

25

25

Version 3 in Action

Same as Version 2

26

26

Makefile Gotchas



Beware:

- Each command (i.e., second line of each dependency rule) must begin with a tab character, not spaces
- Use the `rm -f` command with caution

27

27

Making Makefiles



In this course

- Create Makefiles manually

Beyond this course

- Can use tools to generate Makefiles
 - See `mkmf`, others

28

28

Advanced: Automatic Variables



`make` has wildcard matching for generalizing rules

- `make` has "pattern" rules that use % in targets and dependencies
- `make` has variables to fill in the "pattern" in commands
 - `$(@)` : the target of the rule that was triggered
 - `$(<)` : the first dependency of the rule
 - `$(?)` : all the dependencies that are newer than the target
 - `$(^)` : all the dependencies

Examples:

```
testintmath: testintmath.o intmath.o
    $(CC) $(CFLAGS) $^ -o $@
%.o: %.c intmath.h
    $(CC) $(CFLAGS) -c $<
```

Not required (and potentially confusing!), but common.

29

29

Advanced: Implicit Rules



`make` has implicit rules for compiling and linking C programs

- `make` knows how to build `x.o` from `x.c`
 - Automatically uses `$(CC)` and `$(CFLAGS)`
- `make` knows how to build an executable from `.o` files
 - Automatically uses `$(CC)`

`make` has implicit rules for inferring dependencies

- `make` will assume that `x.o` depends upon `x.c`

Not required (and potentially confusing):
see appendix of these slides for details!

30

30

Make Resources



C Programming: A Modern Approach (King) Section 15.4

GNU `make`

- <http://www.gnu.org/software/make/manual/make.html>

31

31

Summary



Motivation for Make

- Automation of partial builds

Make fundamentals (Makefile version 1)

- Dependency rules, targets, dependencies, commands

Non-file targets (Makefile version 2)

Macros (Makefile version 3)

32

32



Debugging (Part 1)



The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 5

33

33

Goals of this Lecture



Help you learn about:

- Strategies and tools for debugging your code

Why?

- Debugging large programs can be difficult
- A power programmer knows a wide variety of debugging **strategies**
- A power programmer knows about **tools** that facilitate debugging
 - Debuggers
 - Version control systems

34

34

Testing vs. Debugging



Testing

- What should I do to try to **break** my program?

Debugging

- What should I do to try to **fix** my program?

35

35

Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for familiar bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

36

36

Understand Error Messages



Debugging at **build-time** is easier than
debugging at **run-time**, if and only if you...

Understand the error messages!

```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0.
int main(void)
{ printf("hello, world\n");
  return 0;
}
```

What are the errors? (No fair looking at the next slide!)

37

37

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0.
int main(void)
{ printf("hello, world\n");
  return 0;
}
```

Which tool (preprocessor, compiler, or linker) reports the error(s)?

```
$ gcc217 hello.c -o hello
hello.c:1:20: error: stdio.h: No such file or
directory
hello.c:2:1: error: unterminated comment
```

38

38

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
return 0. */
int main(void)
{ printf("hello, world\n");
  return 0;
}
```

What are the errors? (No fair looking at the next slide!)

39

39

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
return 0. */
int main(void)
{ printf("hello, world\n");
  return 0;
}
```

Which tool (preprocessor, compiler, or linker) reports the error?

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:6:4: error: expected ';' before 'return'
hello.c:7:1: warning: control reaches end of non-void function
```

40

40

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
return 0. */
int main(void)
{ printf("hello, world\n");
  return 0;
}
```

What are the errors? (No fair looking at the next slide!)

41

41

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
return 0. */
int main(void)
{ printf("hello, world\n");
  return 0;
}
```

Which tool (preprocessor, compiler, or linker) reports the error?

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:5:1: warning: implicit declaration of function 'printf'
/tmp/ccLSPMTR.o: In function 'main':
hello.c:(.text+0x10): undefined reference to 'printf'
collect2: ld returned 1 exit status
```

42

42

Understand Error Messages



```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{ enum StateType
  { STATE_REGULAR,
    STATE_INWORD
  }
  printf("just hanging around\n");
  return EXIT_SUCCESS;
}
```

What are the errors? (No fair looking at the next slide!)

43

43

Understand Error Messages



```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{ enum StateType
  { STATE_REGULAR,
    STATE_INWORD
  }
  printf("just hanging around\n");
  return EXIT_SUCCESS;
}
```

What does this error message even mean?

```
$ gcc217 hello.c -o hello
hello.c:9:11: error: expected declaration specifiers or '...' before string constant
```

44

44

Understand Error Messages

Caveats concerning error messages

- Line # in error message may be approximate
- Error message may seem nonsensical
- Compiler may not report the real error

Tips for eliminating error messages

- Clarity facilitates debugging
 - Make sure code is indented properly
- Look for missing "punctuation"
 - ; at ends of structure and enum type definitions
 - ; at ends of function declarations
 - ; at ends of do-while loops
- Work incrementally
 - Start at first error message
 - Fix, rebuild, repeat

45

45

Agenda

- (1) Understand error messages
- (2) Think before writing**
- (3) Look for familiar bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

46

46

Think Before Writing

Inappropriate changes could make matters worse, so...

Think before changing your code

- Explain the code to:
 - Yourself
 - Someone else
 - A Teddy bear / plushie stuffed tiger?
- Do experiments
 - But make sure they're disciplined



47

47

Agenda

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs**
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

48

48

Look for Common Bugs

Some of our favorites:

```

switch (i)
{
    case 0:
        ...
        break;
    case 1:
        ...
    case 2:
        ...
}

int i;
...
scanf("%d", i);

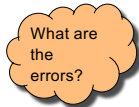
char c;
...
c = getchar();

while (c = getchar() != EOF)
    ...

if (i = 5)
    ...

if (5 < i < 10)
    ...

if (i & j)
    ...
  
```



49

49

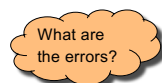
Look for Common Bugs

Some of our favorites:

```

for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; i++)
    {
        ...
    }
}

for (i = 0; i < 10; i++)
{
    for (j = 10; j >= 0; j++)
    {
        ...
    }
}
  
```



50

50

Look for Common Bugs

Some of our favorites:

```
{ int i;
...
i = 5;
if (something)
{ int i;
...
i = 6;
...
}
...
printf("%d\n", i);
...
}
```

What value is written if this statement is present? Absent?

51

51

Agenda

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer**
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

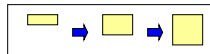
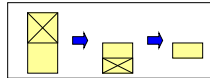
52

52

Divide and Conquer

Divide and conquer: To debug a **program**...

- Incrementally find smallest **input file** that illustrates the bug
- Approach 1: **Remove** input
 - Start with file
 - Incrementally remove lines until bug disappears
 - Examine most-recently-removed lines
- Approach 2: **Add** input
 - Start with small subset of file
 - Incrementally add lines until bug appears
 - Examine most-recently-added lines



53

53

Divide and Conquer

Divide and conquer: To debug a **module**...

- Incrementally find smallest **client subset** that illustrates the bug
- Approach 1: **Remove** code
 - Start with test client
 - Incrementally remove lines of code until bug disappears
 - Examine most-recently-removed lines
- Approach 2: **Add** code
 - Start with minimal client
 - Incrementally add lines of test client until bug appears
 - Examine most-recently-added lines

54

54

Agenda

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests**
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

55

55

Add More Internal Tests

- (5) Add more internal tests**
 - Internal tests help **find** bugs (see "Testing" lecture)
 - Internal test also can help **eliminate** bugs
 - Validating parameters & checking invariants can eliminate some functions from the bug hunt

56

56

Agenda

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output**
- (7) Use a debugger
- (8) Focus on recent changes

57

Display Output

Write values of important variables at critical spots

- Poor:


```
printf("%d", keyvariable);
```

stdout is buffered; program may crash before output appears
- Maybe better:


```
printf("%d\n", keyvariable);
```

Printing '\n' flushes the stdout buffer, but not if stdout is redirected to a file
- Better:


```
printf("%d", keyvariable);
fflush(stdout);
```

Call fflush() to flush stdout buffer explicitly

58

Display Output

- Maybe even better:


```
fprintf(stderr, "%d", keyvariable);
```

Write debugging output to stderr; debugging output can be separated from normal output via redirection
- Maybe better still:


```
FILE *fp = fopen("logfile", "w");
...
fprintf(fp, "%d", keyvariable);
fflush(fp);
```

Bonus: stderr is unbuffered
Write to a log file

59

Agenda

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger**
- (8) Focus on recent changes

60

Use a Debugger

Use a debugger

- Alternative to displaying output

61

The GDB Debugger

GNU Debugger

- Part of the GNU development environment
- Integrated with Emacs editor
- Allows user to:
 - Run program
 - Set breakpoints
 - Step through code one line at a time
 - Examine values of variables during run
 - Etc.

For details see precept tutorial, precept reference sheet, Appendix 2 of these slides

62

Agenda

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) **Focus on recent changes**

63

Focus on Recent Changes

Focus on recent changes

- Corollary: Debug now, not later

Difficult:

- (1) Compose entire program
- (2) Test entire program
- (3) Debug entire program

Easier:

- (1) Compose a little
- (2) Test a little
- (3) Debug a little
- (4) Compose a little
- (5) Test a little
- (6) Debug a little
- ...

64

Focus on Recent Changes

Focus on recent change (cont.)

- Corollary: Maintain old versions

Difficult:

- (1) Change code
- (2) Note new bug
- (3) Try to remember what changed since last version

Easier:

- (1) Backup current version
- (2) Change code
- (3) Note new bug
- (4) Compare code with last version to determine what changed

65

Maintaining Old Versions

To maintain old versions...

Approach 1: Manually copy project directory

```
...
$ mkdir myproject
$ cd myproject

    Create project files here.

$ cd ..
$ cp -r myproject myprojectDateTime
$ cd myproject

    Continue creating project files here.

...
```

66

Maintaining Old Versions

Approach 2: Use a Revision Control System such as subversion or git

- Allows programmer to:
 - **Check-in** source code files from **working copy** to **repository**
 - **Commit** revisions from **working copy** to **repository**
 - saves all old versions
 - **Update** source code files from **repository** to **working copy**
 - Can retrieve old versions
- Appropriate for one-developer projects
- Extremely useful, almost *necessary* for multideveloper projects!

Not required for COS 217, but good to know!

Google "subversion svn" or "git" for more information.

67

Summary

General debugging strategies and tools:

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
 - Use GDB!!!
- (8) Focus on recent changes
 - Consider using git, etc.

68

Appendix 1: Implicit Rules

make has implicit rules for compiling and linking C programs

- **make** knows how to build x.o from x.c
 - Automatically uses \$(CC) and \$(CFLAGS)
- **make** knows how to build an executable from .o files
 - Automatically uses \$(CC)

```
intmath.o: intmath.c intmath.h
$(CC) $(CFLAGS) -c intmath.c
```

```
intmath.o: intmath.c intmath.h
```

```
testintmath: testintmath.o intmath.o
$(CC) testintmath.o intmath.o -o testintmath
```

```
testintmath: testintmath.o intmath.o
```

69

Makefile Version 4

```
# Macros
CC = gcc217
# CC = gcc217m
CFLAGS =
# CFLAGS = -g
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O

# Dependency rules for non-file targets
all: testintmath
clobber: clean
rm -f *~ \#*\#
clean:
rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
testintmath.o: testintmath.c intmath.h
intmath.o: intmath.c intmath.h
```

70

69

70

Version 4 in Action

Same as Version 2

71

71

Implicit Dependencies

make has implicit rules for inferring dependencies

- **make** will assume that x.o depends upon x.c

```
intmath.o: intmath.c intmath.h
```

```
intmath.o: intmath.h
```

72

72

Makefile Version 5

```
# Macros
CC = gcc217
# CC = gcc217m
CFLAGS =
# CFLAGS = -g
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O

# Dependency rules for non-file targets
all: testintmath
clobber: clean
rm -f *~ \#*\#
clean:
rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
testintmath.o: intmath.h
intmath.o: intmath.h
```

73

73

Version 5 in Action

Same as Version 2

74

74

Makefile Gotchas

Beware:

- To use an implicit rule to make an *executable*, the executable must have the same name as one of the .o files

Correct: `myprog: myprog.o someotherfile.o` ✓

Won't work: `myprog: somefile.o someotherfile.o` ✗

75

Appendix 2: Using GDB

An example program

File testintmath.c:

Euclid's algorithm;
Don't be concerned
with details

```
#include <stdio.h>

int gcd(int i, int j)
{
    int temp;
    while (j != 0)
    {
        temp = i % j;
        i = j;
        j = temp;
    }
    return i;
}

int lcm(int i, int j)
{
    return (i / gcd(i, j)) * j;
}

int main(void)
{
    int iGcd;
    int iLcm;
    iGcd = gcd(8, 12);
    iLcm = lcm(8, 12);
    printf("%d %d\n", iGcd, iLcm);
    return 0;
}
```

The program is correct

But let's pretend it has a
runtime error in `gcd()`...

76

Using GDB

General GDB strategy:

- Execute the program to the point of interest
 - Use breakpoints and stepping to do that
- Examine the values of variables at that point

77

Using GDB

Typical steps for using GDB:

- Build with `-g`

```
gcc217 -g testintmath.c -o testintmath
```

 - Adds extra information to executable file that GDB uses
- Run Emacs, with no arguments


```
emacs
```
- Run GDB on executable file from within Emacs


```
<Esc key> x gdb <Enter key> testintmath <Enter key>
```
- Set breakpoints, as desired


```
break main
```

 - GDB sets a breakpoint at the first executable line of `main()`

```
break gcd
```

 - GDB sets a breakpoint at the first executable line of `gcd()`

78

Using GDB

Typical steps for using GDB (cont.):

(e) Run the program

```
run
```

- GDB stops at the breakpoint in `main()`
- Emacs opens window showing source code
- Emacs highlights line that is to be executed next

```
continue
```

- GDB stops at the breakpoint in `gcd()`
- Emacs highlights line that is to be executed next

(f) Step through the program, as desired

```
step (repeatedly)
```

- GDB executes the next line (repeatedly)

- Note: When next line is a call of one of your functions:

- `step` command *steps into* the function
- `next` command *steps over* the function, that is, executes the next line without stepping into the function

79

Using GDB

Typical steps for using GDB (cont.):

- Examine variables, as desired


```
print i
print j
print temp
```

 - GDB prints the value of each variable
- Examine the function call stack, if desired


```
where
```

 - GDB prints the function call stack
 - Useful for diagnosing crash in large program
- Exit gdb


```
quit
```
- Exit Emacs


```
<Ctrl-x key> <Ctrl-c key>
```

80

Using GDB



GDB can do much more:

- Handle command-line arguments
`run arg1 arg2`
- Handle redirection of stdin, stdout, stderr
`run < somefile > someotherfile`
- Print values of expressions
- Break conditionally
- Etc.

81

81